University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

5 YEARS

Pitt SCI

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 8: this Friday @ 11:59 pm
  - Assignment 2: this Friday @ 11:59 pm
    - Support video and slides on Canvas
  - Lab 7: Tuesday 3/21 @ 11:59 pm

Previous lecture

- LZW example and corner case
- Shannon’s Entropy
- LZW vs. Huffman
- Burrows-Wheeler Compression Algorithm

This Lecture

- Burrows-Wheeler Compression Algorithm
- ADT Graph
  - definitions
  - representations

Burrows-Wheeler Data Compression Algorithm

- **Best** compression algorithm (in terms of compression ratio) **for text**
- The basis for UNIX's **bzip2** tool

Adapted from: https://www.cs.princeton.edu/courses/archive/spr03/cos226/assignments/burrows.html

BWT: Compression Algorithm

- Three steps
  - Burrows-Wheeler Transform
    - Cluster same letters as close to each other as possible
  - Move-To-Front Encoding
    - Convert output of previous step into an integer file with **large frequency** differences
  - Huffman Compression
    - Compress the file of integers using Huffman Compression

BWT: Expansion Algorithm

- Apply the inverse of compression steps in reverse order
  - Huffman decoding
  - Move-To-Front decoding
  - Inverse Burrows-Wheeler Transform

Move-To-Front Encoding

- Initialize an ordered list of the 256 ASCII characters
  - character *i* appears *i*th in the list
- For each character c from input
  - output the index in the list where c appears
  - move c to the front of the list (i.e., index 0)

Move-To-Front Encoding

Input:

e a e d e e

0	**a**
1	**b**
2	**c**
3	**d**
4	**e**

Output:

Move-To-Front Encoding

Input:

e a e d e e

0	a
1	b
2	c
3	d
4	e

Output:

4

Move-To-Front Encoding

Input:

e a e d e e

0	**a**	**e**
1	**b**	**a**
2	**c**	**b**
3	**d**	**c**
4	**e**	**d**

Output:

4

Move-To-Front Encoding

Input:

e a e d e e

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

4 1

Move-To-Front Encoding

-

Input:

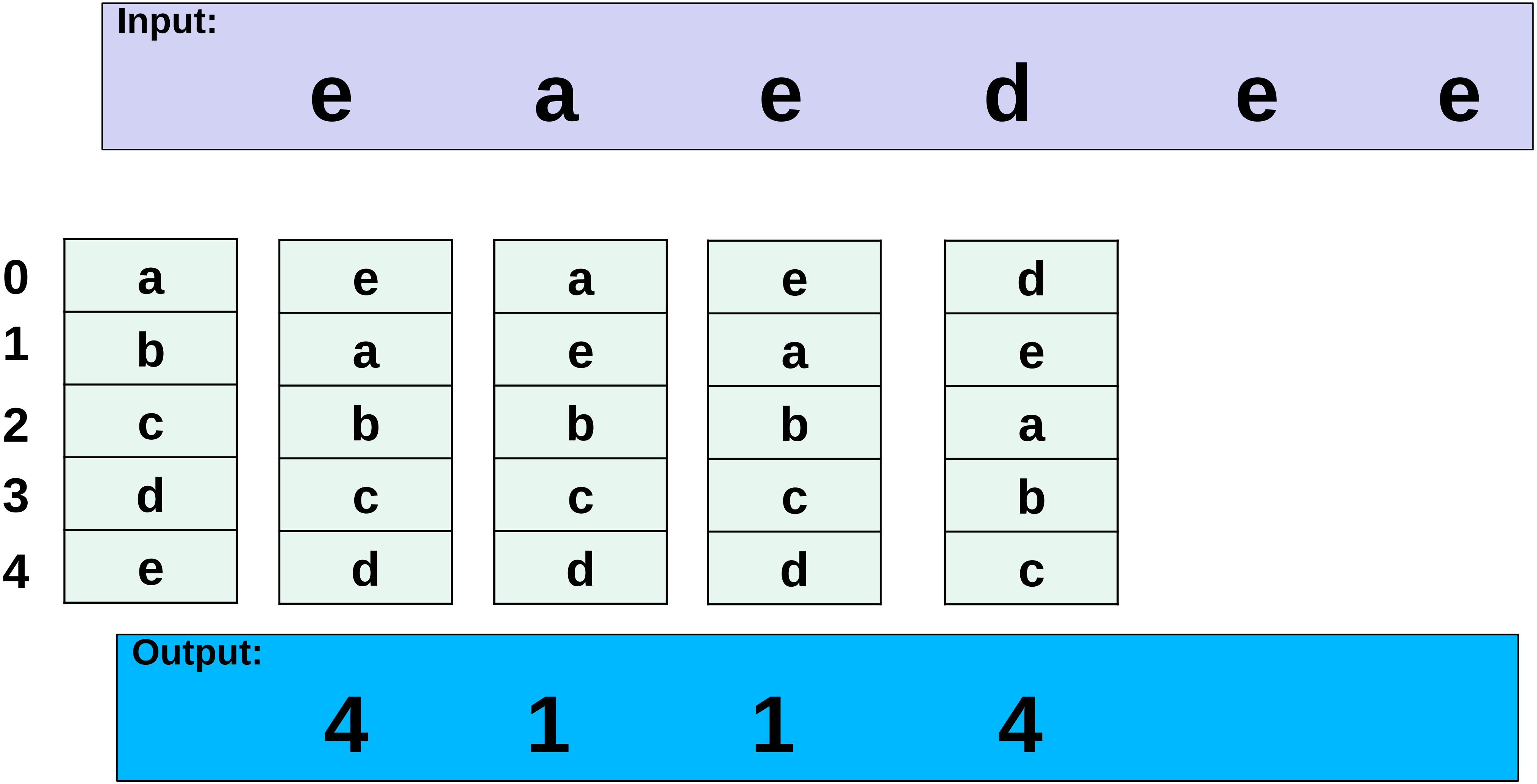
e a e d e e

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

4 1 1

Move-To-Front Encoding



Input:

e a e d e e

0	a	e	a	e	d
1	b	a	e	a	e
2	c	b	b	b	a
3	d	c	c	c	b
4	e	d	d	d	c

Output:

4 1 1 4

Move-To-Front Encoding

Input:

e a e d e e

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

4 1 1 4 1

Move-To-Front Encoding

Input:

e a e d e e

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

4 1 1 4 1 0

Move-To-Front Encoding

In the output of MTF Encoding, smaller integers have higher frequencies than larger integers

Move-To-Front Decoding

- Initialize an ordered list of 256 characters
  - same as encoding
- For each integer *i* (*i* is between 0 and 255)
  - print the *i*th character in the list
  - move that character to the front of the list

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a
1	b
2	c
3	d
4	e

Output:

e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	**a**	**e**
1	**b**	**a**
2	**c**	**b**
3	**d**	**c**
4	**e**	**d**

Output:

e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

e a

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

e a e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e	d
1	b	a	e	a	e
2	c	b	b	b	a
3	d	c	c	c	b
4	e	d	d	d	c

Output:

e a e d

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

e a e d e

Move-To-Front Decoding

- **Decoding**

Input:

4 1 1 4 1 0

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

e a e d e e

BWT: Compression Algorithm

- **Compression**
  - Burrows-Wheeler Transform
  - Move-To-Front Encoding ✓
  - Huffman Compression ✓

- **Expansion**
  - Huffman decoding ✓
  - Move-To-Front decoding ✓
  - Inverse Burrows-Wheeler Transform

Burrows-Wheeler Transform

- **Rearranges** the characters in the input
  - lots of clusters with **repeated characters**
  - still possible to **recover** the original input
- Intuition: Consider the string **hen** in English text
  - most of the time the letter preceding it is t or w
  - group all such preceding letters together (mostly t's and some w’s)

Burrows-Wheeler Transform

- For each block of length N characters
  - generate **N strings** by **cycling** the characters of the block one step at a time
  - **sort** the strings
  - output is the **last column** in the sorted table and the **index** of the original block in the sorted array

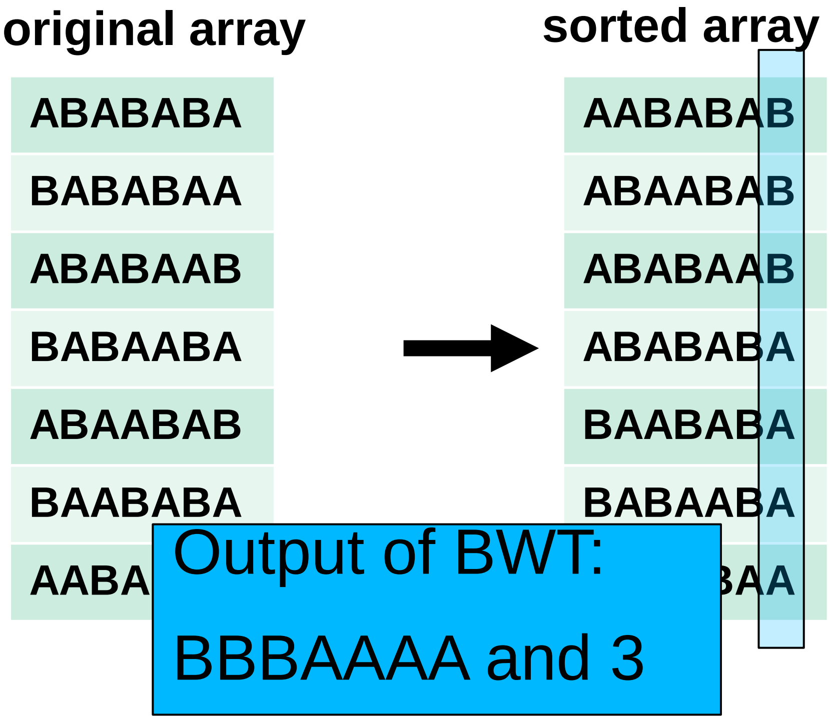
Burrows-Wheeler Transform

- Example: Let’s transform “ABRACADABRA”
- N = 11
- Cyclic Versions of the string:
  - ABRACADABRA
  - BRACADABRAA
  - RACADABRAAB
  - ACADABRAABR
  - CADABRAABRA
  - ADABRAABRAC
  - DABRAABRACA
  - ABRAABRACAD
  - BRAABRACADA
  - RAABRACADAB
  - AABRACADABR
- After Sorting
  - AABRACADABR
  - ABRAABRACAD
  - 2 → ABRACADABRA
  - ACADABRAABR
  - ADABRAABRAC
  - BRAABRACADA
  - BRACADABRAA
  - CADABRAABRA
  - DABRAABRACA
  - RAABRACADAB
  - RACADABRAAB

RDARCAAAABB

Burrows-Wheeler Transform Example 2

- Input: ABABABA
- **Step 1: Build an array of 7 strings, each a circular rotation of the original by one character**
- ABABABA
- BABABAA
- ABABAAB
- BABAABA
- ABAABAB
- BAABABA
- AABABAB
- **Step 2: Sort the array alphabetically**



original array	sorted array
ABABABA	AABABAB
BABABAA	ABAABAB
ABABAAB	ABABAAB
BABAABA	ABABABA
ABAABAB	BAABABA
BAABABA	BABAABA
AABA	BAA

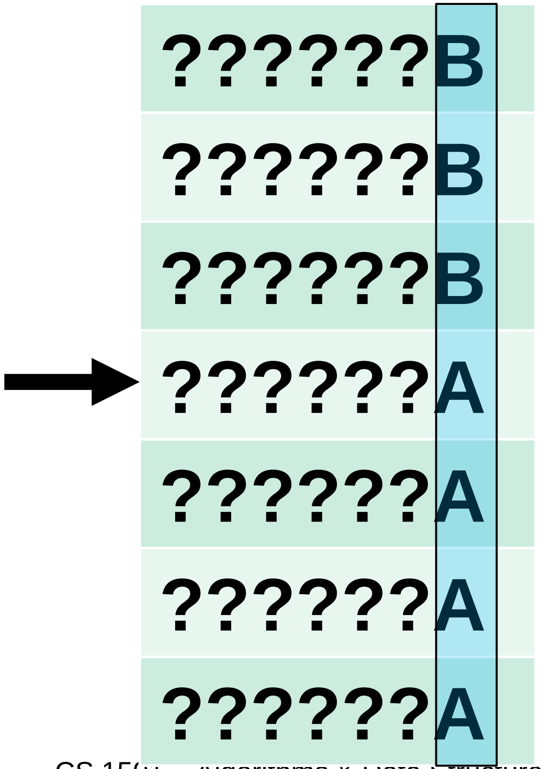
Output of BWT:

BBBAAAA and 3

- **Notice that** the first column of the sorted array has the same characters as the last column
    - all columns have the same set of letters
- **Step 3: Output the last column of the sorted array and the index of the input string in the sorted array**

Burrows-Wheeler Transform Decoding

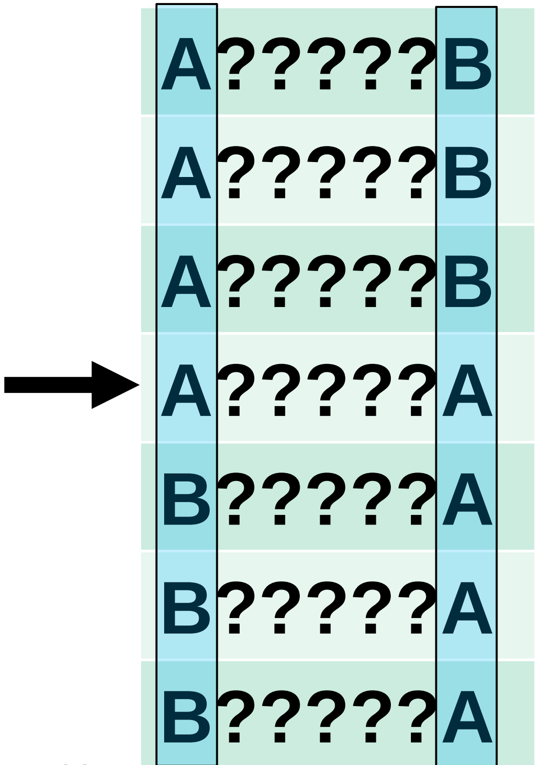
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
  - BBBAAAA → AAAABBB
  - The first column of the sorted array has the same characters as the last column
    - but in sorted order



??????B
??????B
??????B
??????A
??????A
??????A
??????A

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
  - BBBAAAA → AAAABBB
  - This gives us the first column of the sorted array

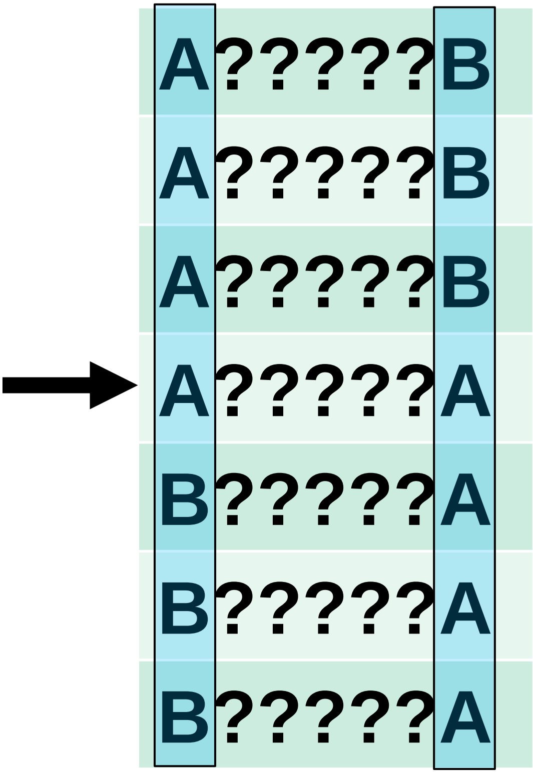


A?????B
A?????B
A?????B
A?????A
B?????A
B?????A
B?????A

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - holds the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column

original array
ABABABA
BABABAA
ABABAAB
BABAABA
ABAABAB
BAABABA
AABABAB



A?????B

A?????B

A?????B

A?????A

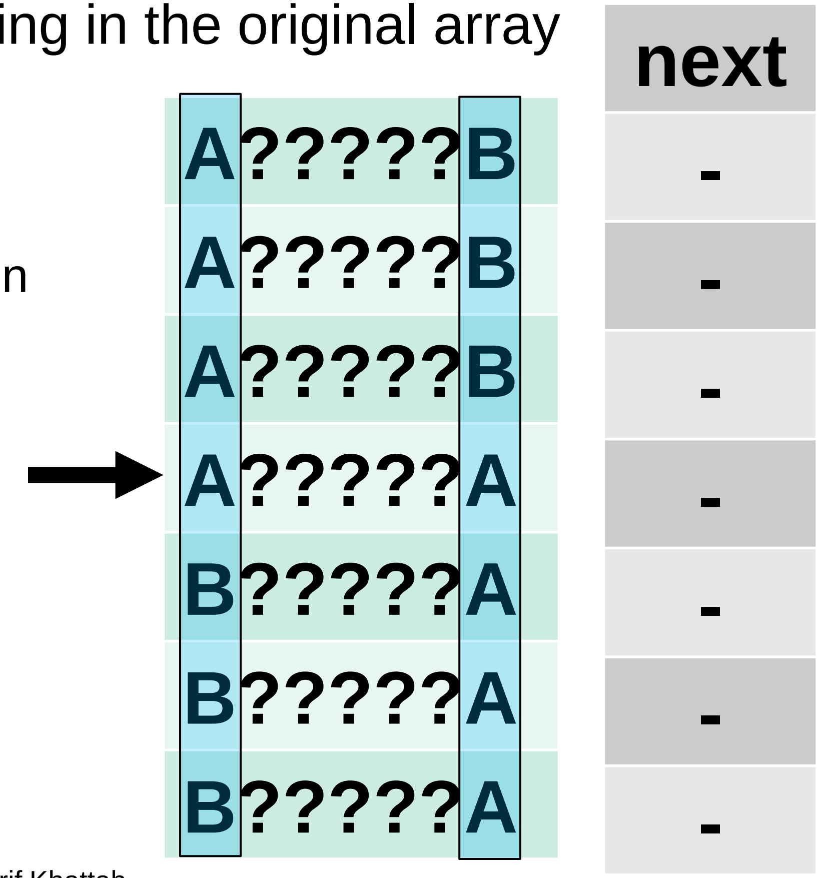
B?????A

B?????A

B?????A

Burrows-Wheeler Transform Decoding

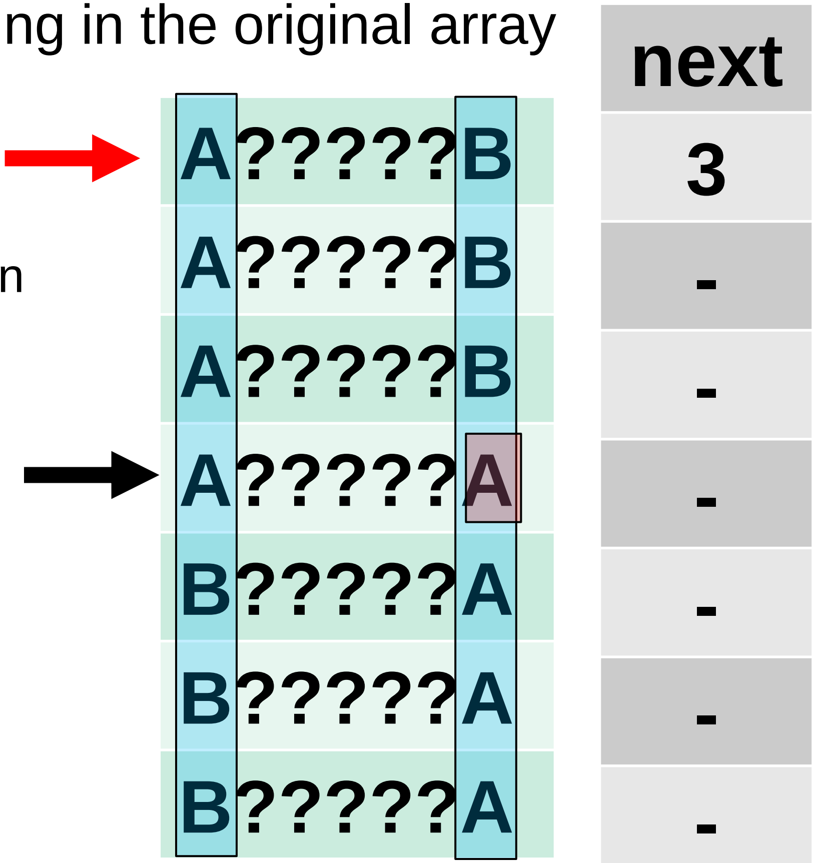
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	-
A?????B	-
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

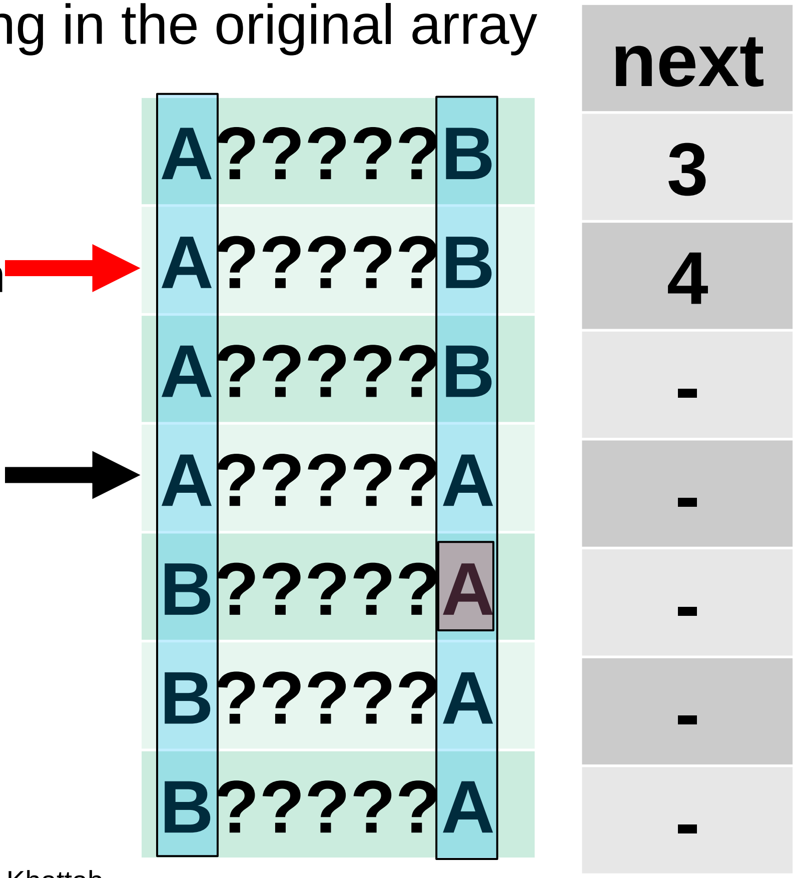
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	-
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

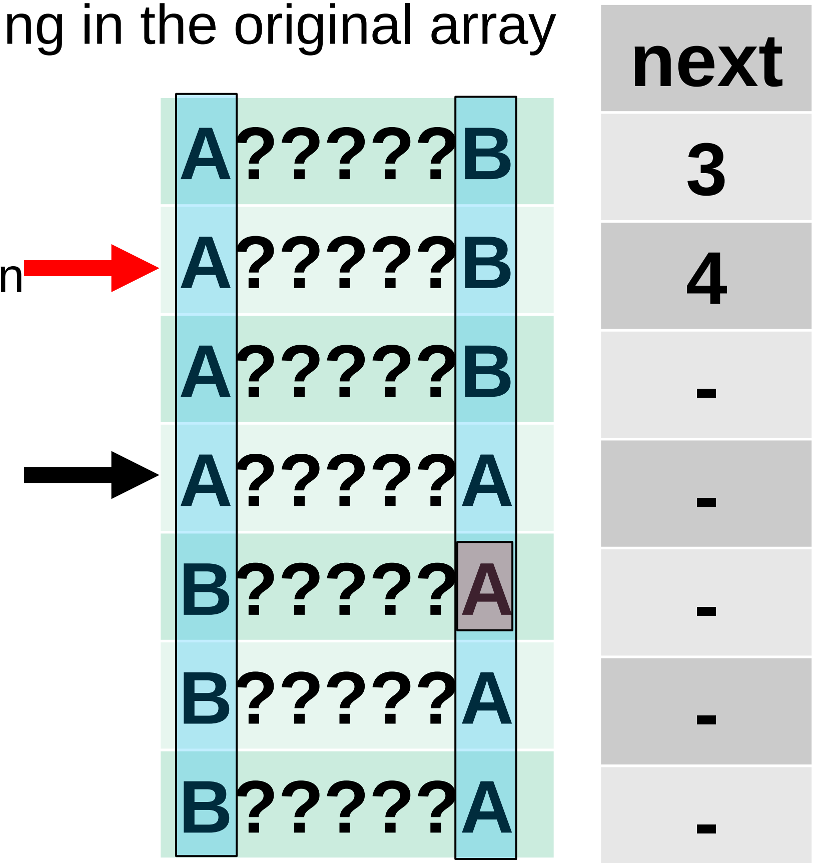
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	4
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

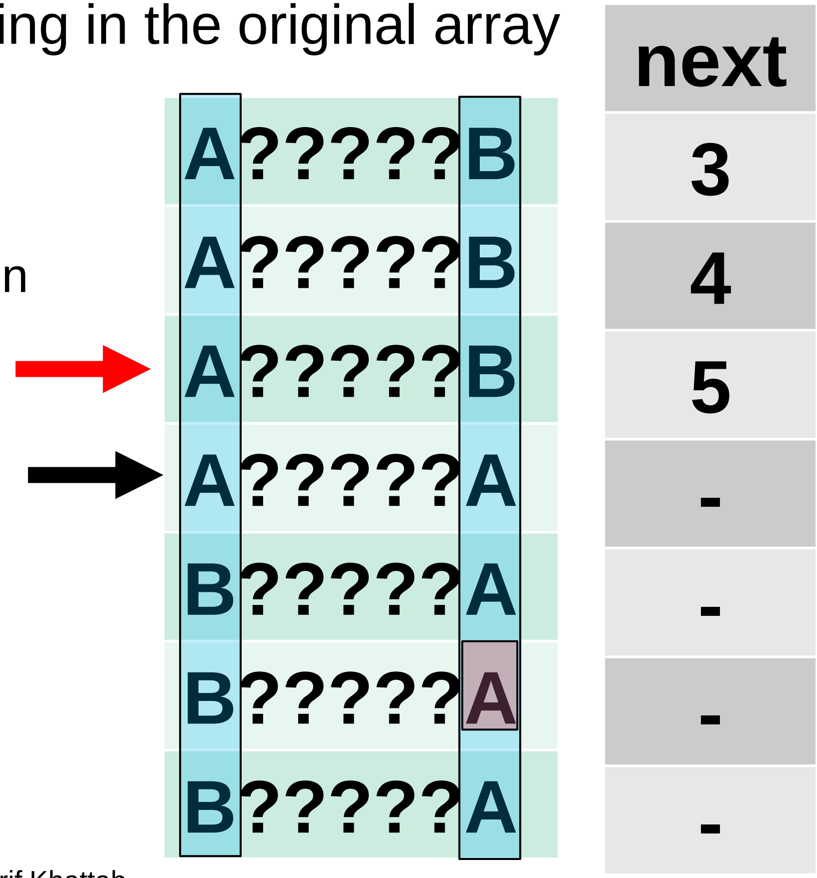
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	4
A?????B	-
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

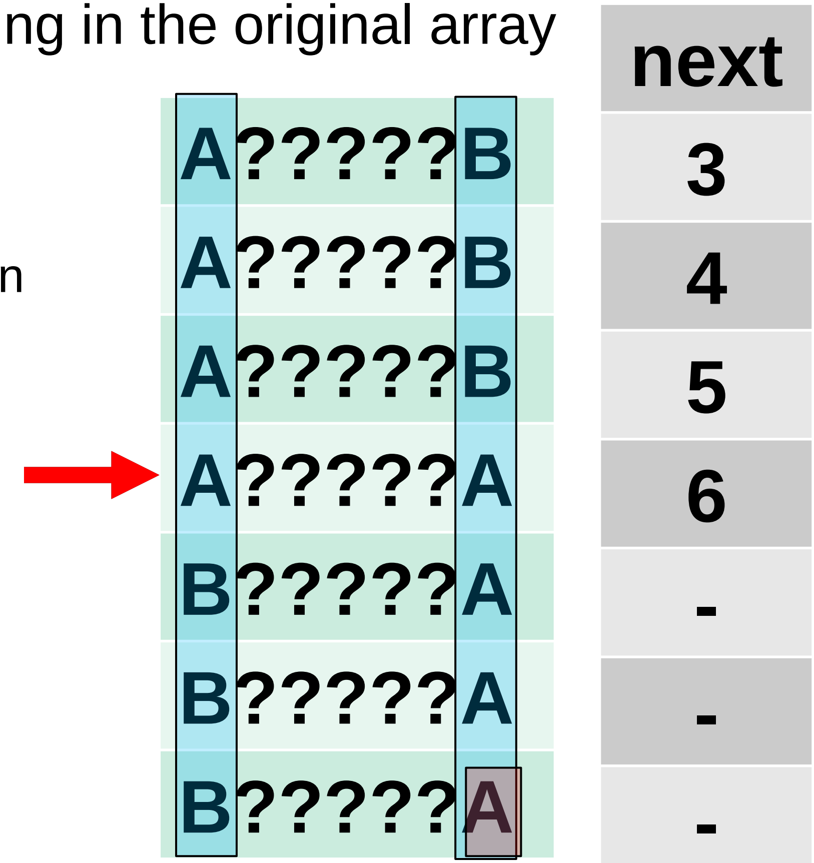
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	-
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

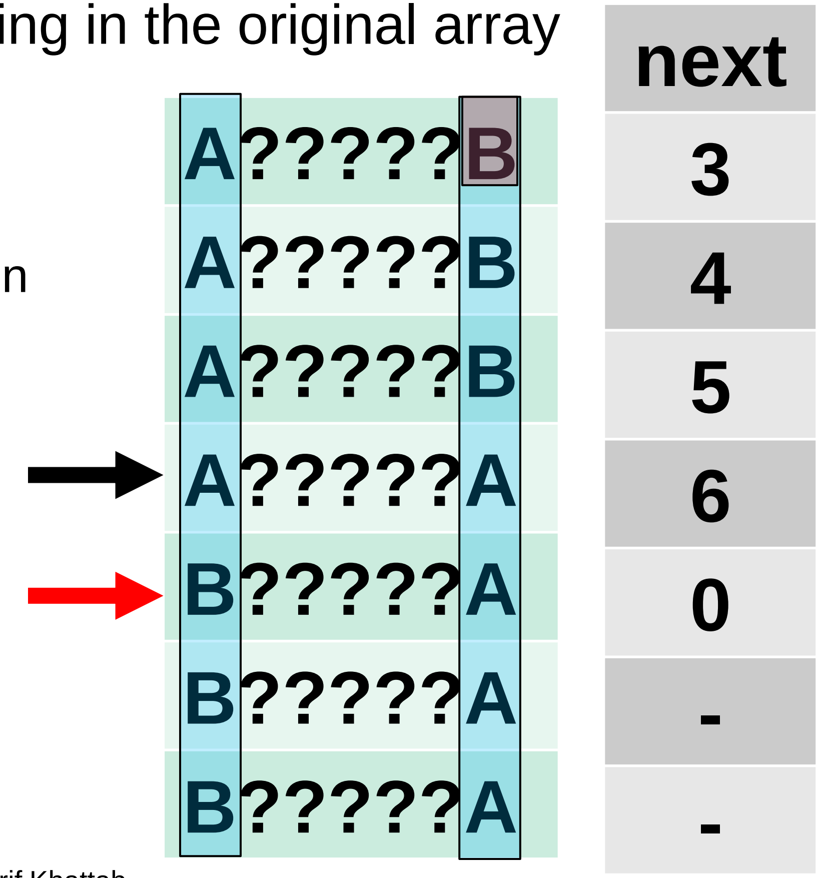
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	-
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

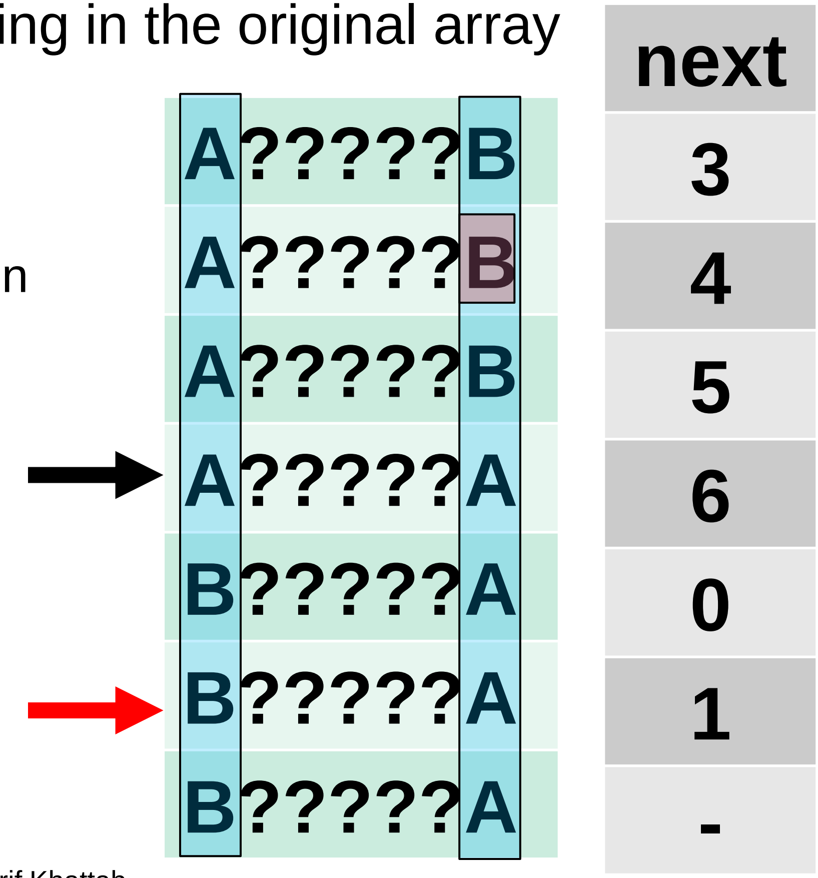
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	-
B?????A	-

Burrows-Wheeler Transform Decoding

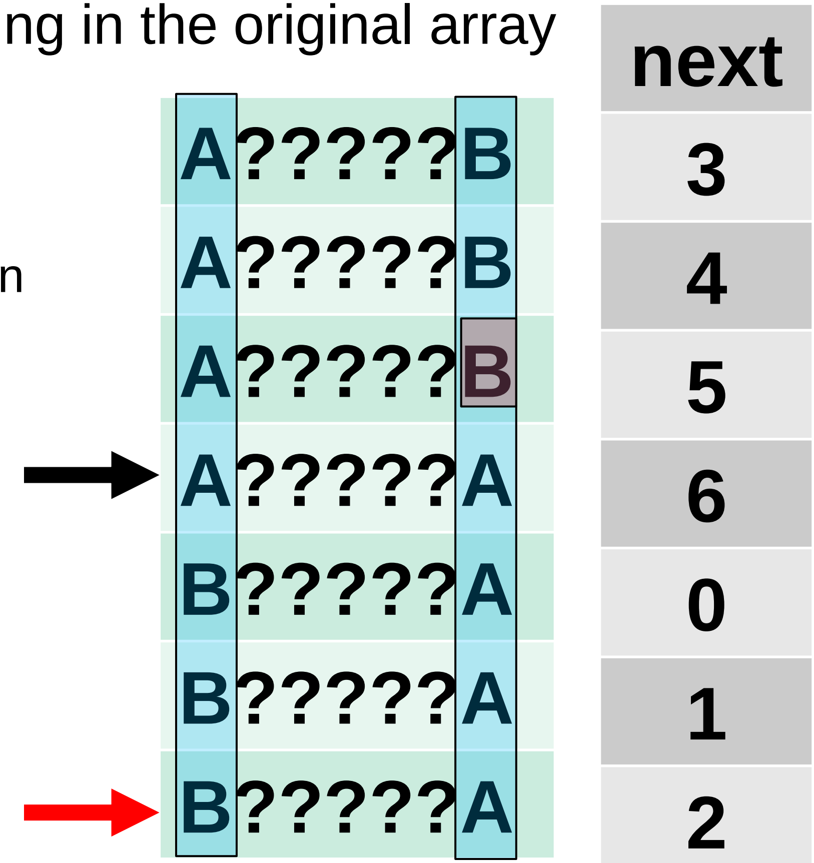
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	-

Burrows-Wheeler Transform Decoding

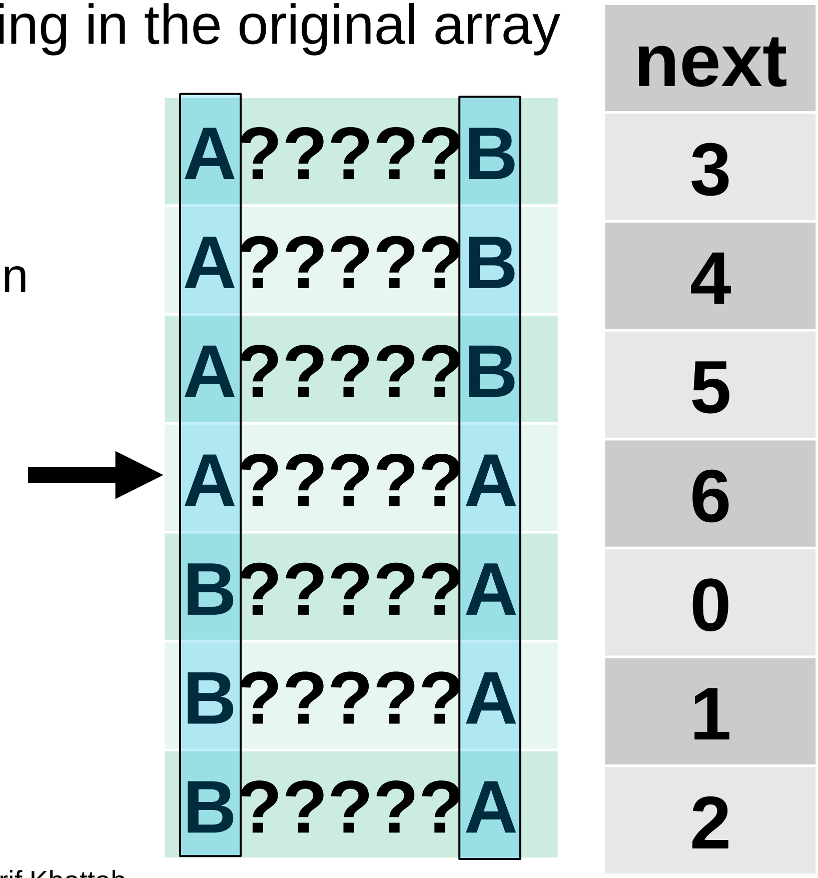
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column
- Why does that work?
  - first character of a string becomes the last character in the next string in the original order

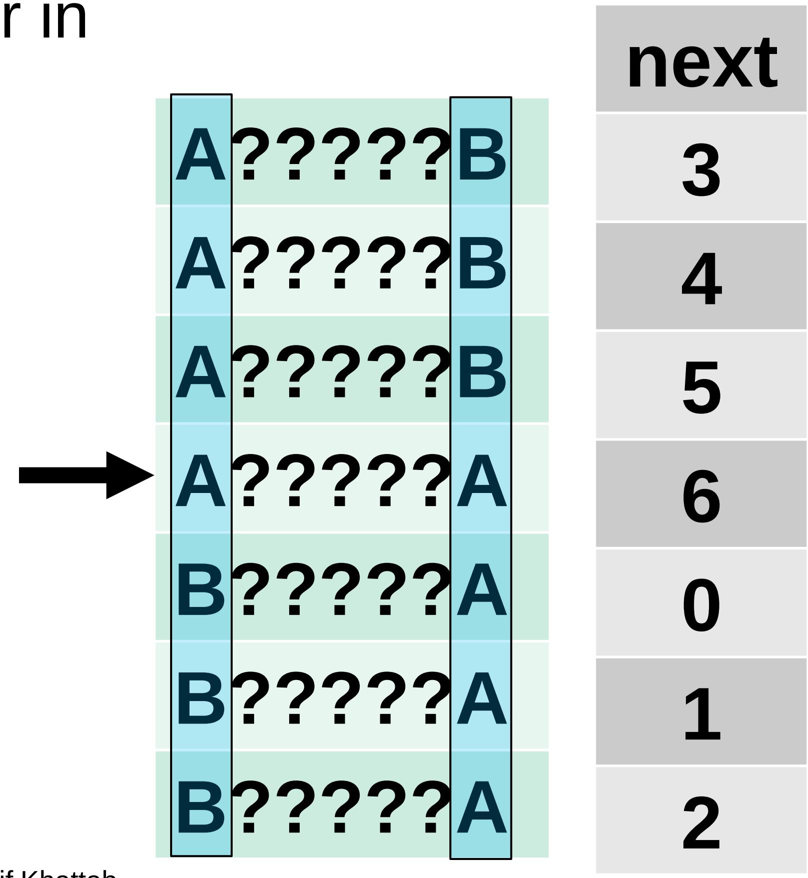


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?

A??????

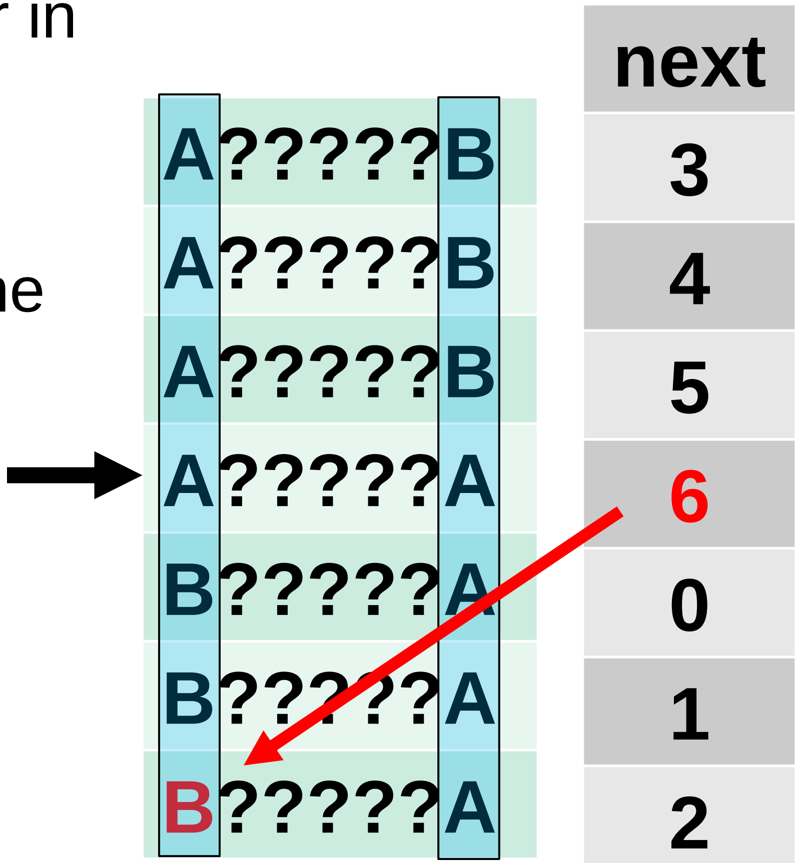


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[3]

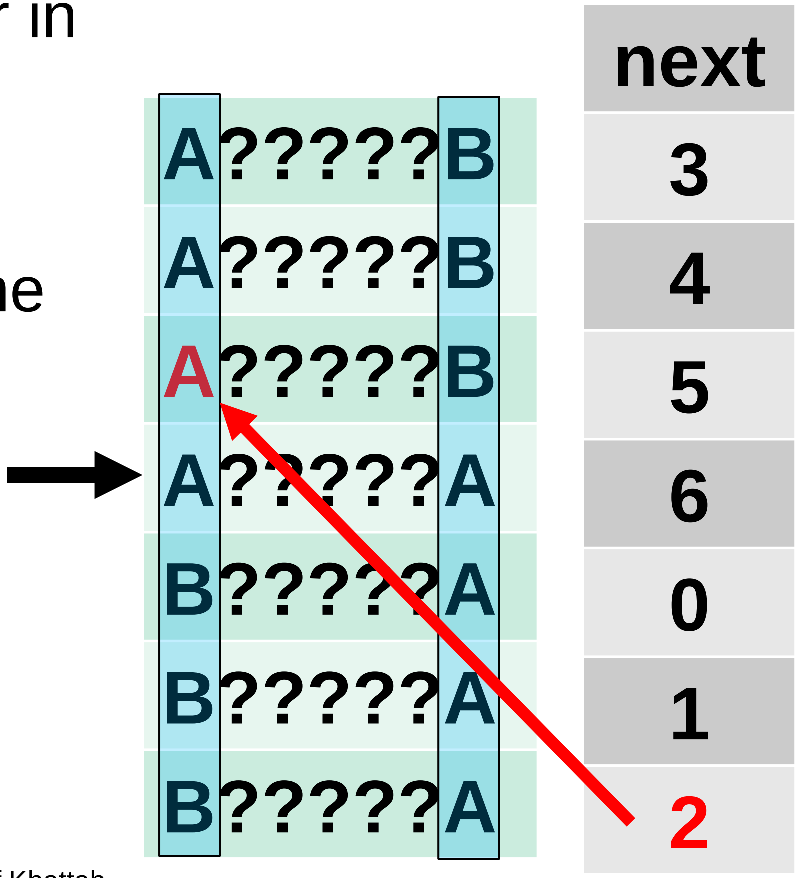
AB?????



Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[6]

ABA????

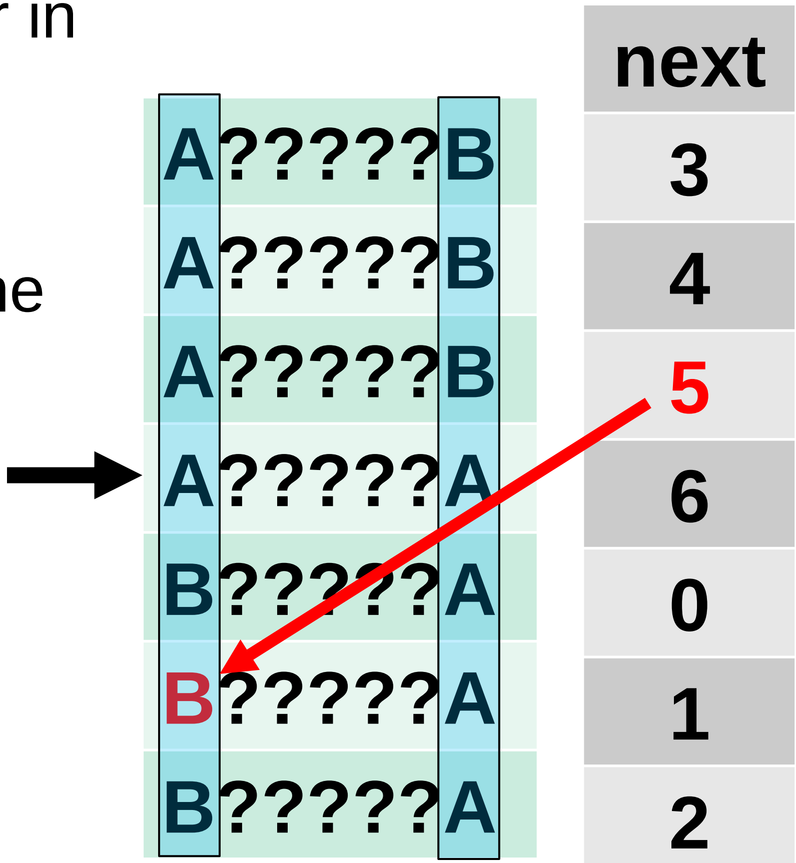


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[2]

ABAB???

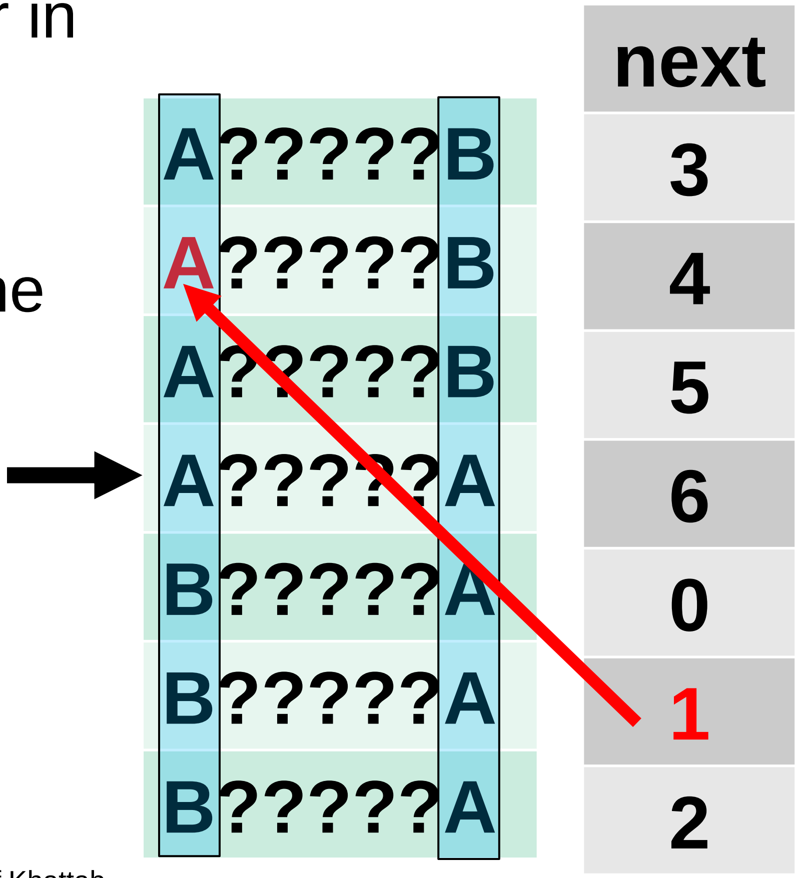


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[5]

ABABA??

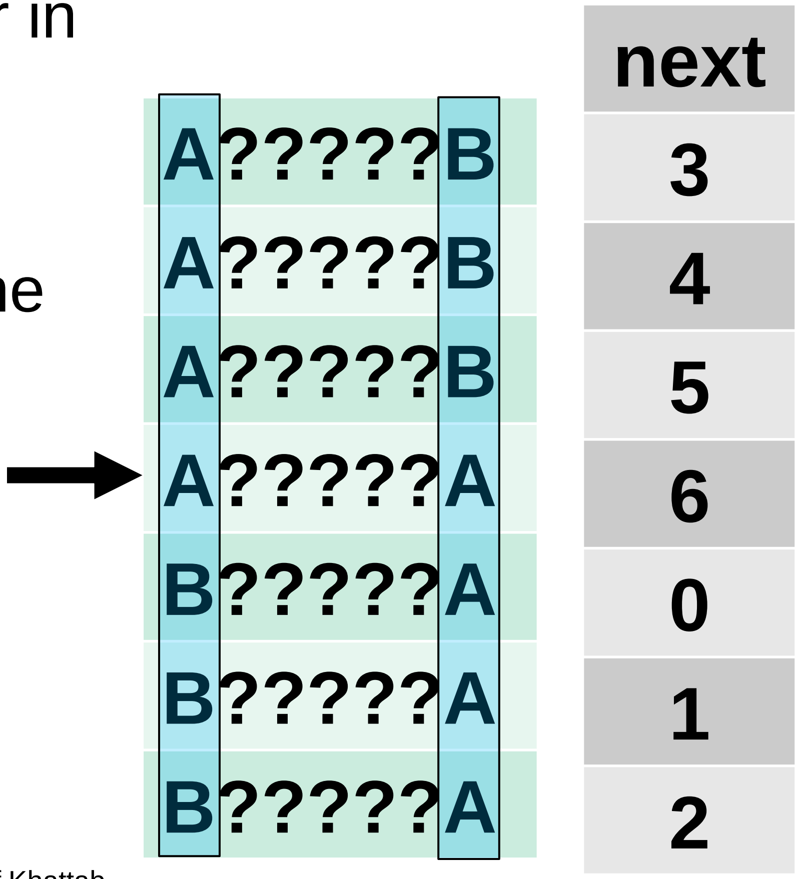


	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[5]

ABABABA



	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

Downsides of Burrows-Wheeler Algorithm

- process **blocks** of input file
  - Compared to LZW, which processes the input **one character at time**
- The **larger** the block size, the **better** the compression
  - But the **longer** the sorting time

A new problem!!

- **Input**: A file containing LinkedIn (LI) accounts and their connections
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

Problem of the Day

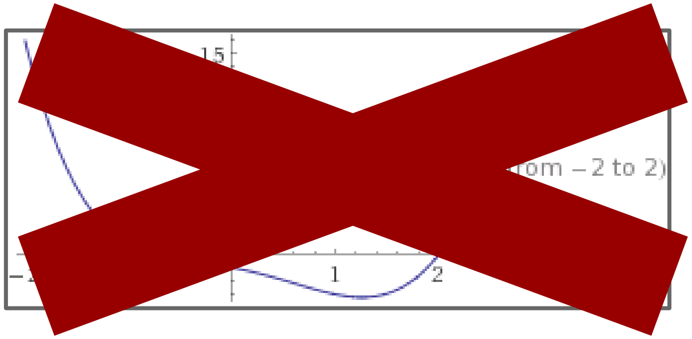
- **Output**: Answer the following questions:
  - Given two LI accounts, how “far” are they from each other?
    - e.g., 1st connection?, 2nd connection?, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - For each connected component, are there certain accounts that if removed, the remaining accounts become ***partitioned***?

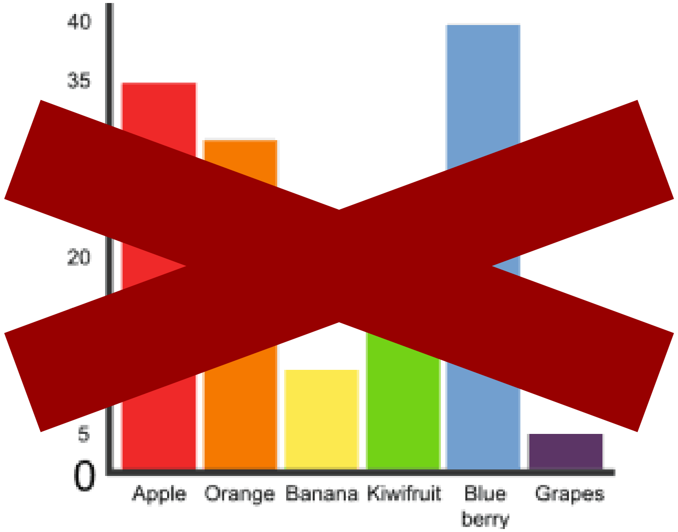
Which Data Type to use?

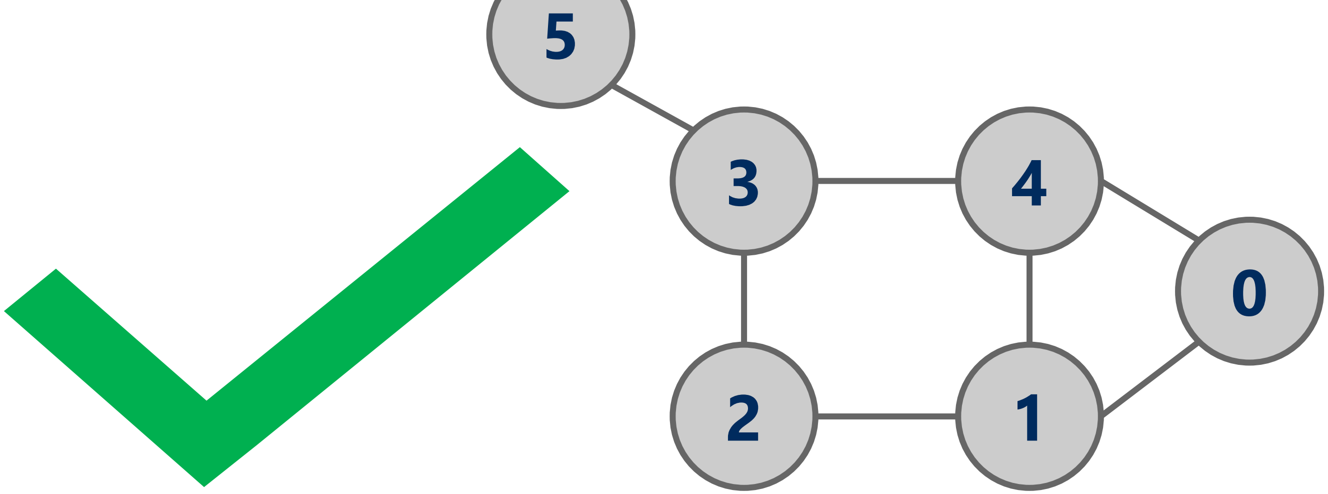
- Let’s think first about how to organize the data that we have in memory
- Note that the operations are different from what we have been used to (search, sort, min, max, add, delete, …)

- Account1: Connection1, Connection2, …
- Account2: Connection1, Connection2, …
- …

Graphs!

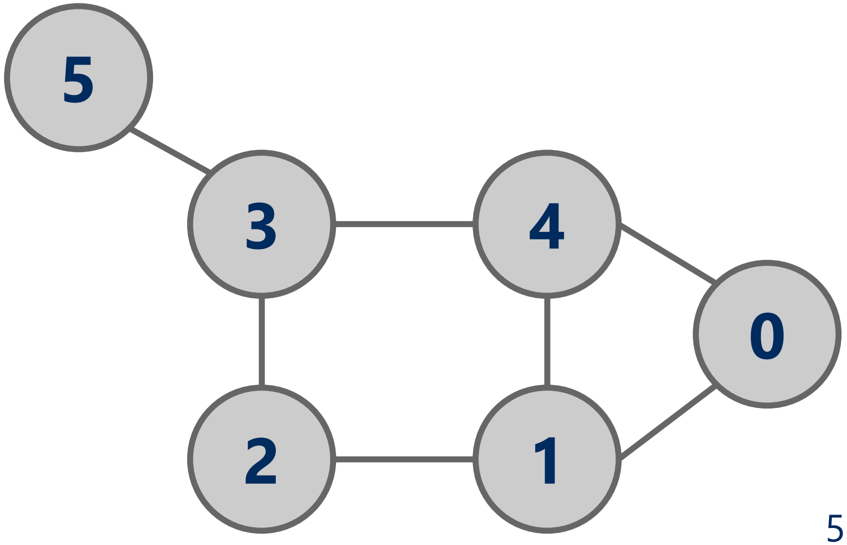

15
rom – 2 to 2)
–2
1
2


40
35
20
5
0
Apple
Orange
Banana
Kiwifruit
Blue berry
Grapes


5
3
4
0
2
1

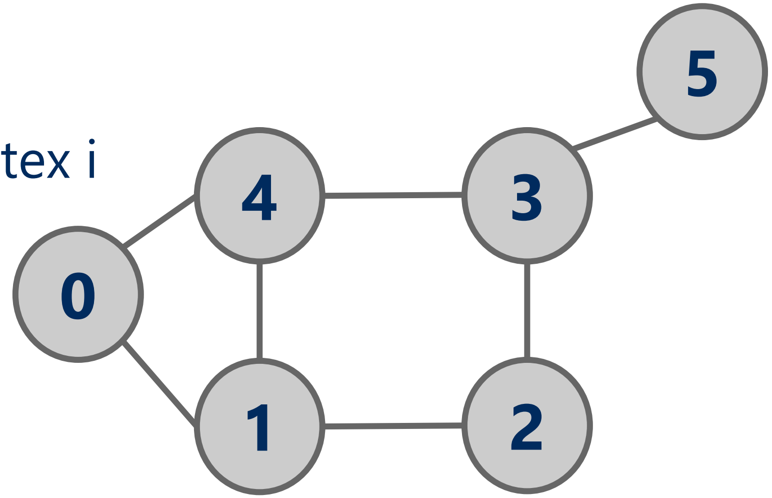
Graphs

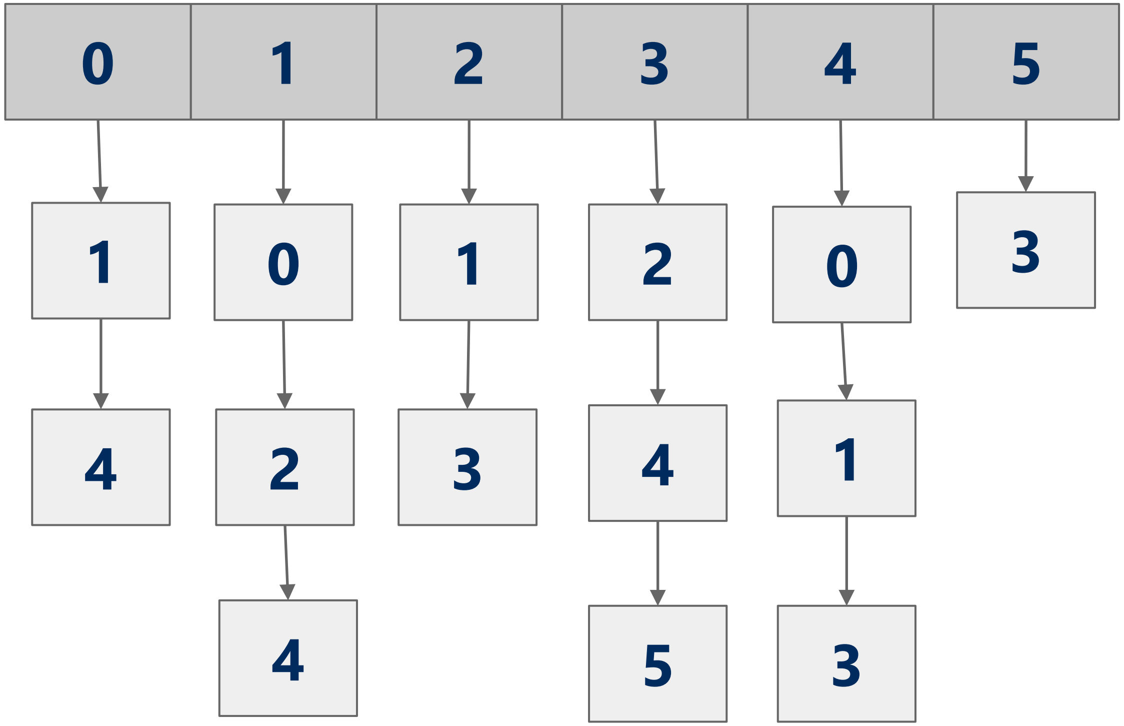
- A graph G = (V, E)
  - where V is a set of vertices
  - E is a set of edges connecting vertex pairs
- Example:
  - V = {0, 1, 2, 3, 4, 5}
  - E = {(0, 1), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4), (3, 5)}



Adjacency lists

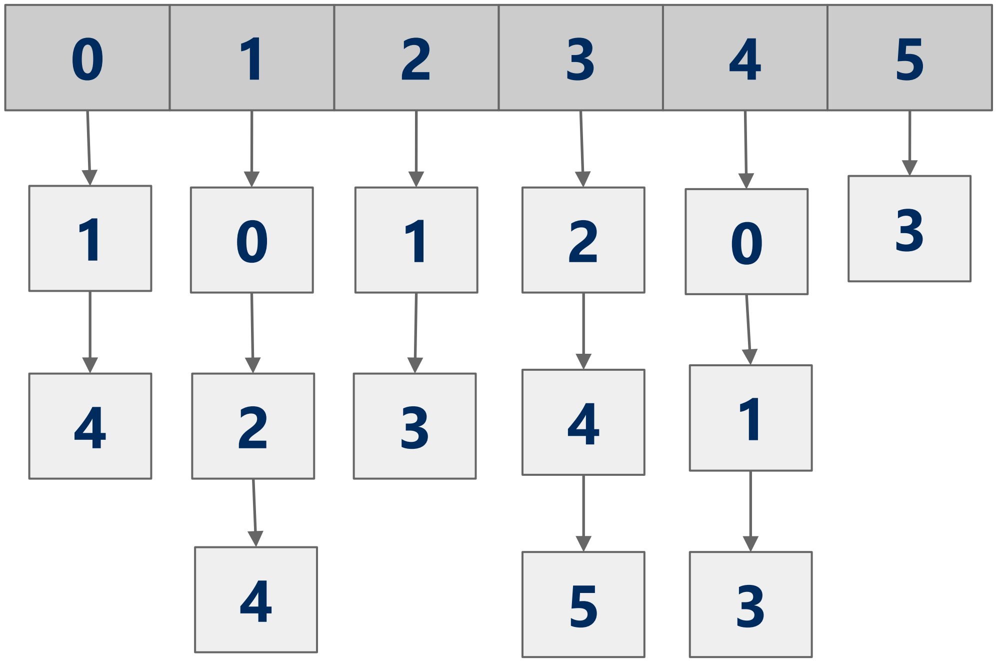
- Array of neighbor lists
  - ○ A[i] contains a list of the neighbors of vertex i





0	1	2	3	4	5
1	0	1	2	0	3
4	2	3	4	1	
	4		5	3	

Adjacency list analysis



- Runtime?
  - Check if two vertices are neighbors
  - Find the list of neighbors of a vertex
    - Θ(d)
    - d is the degree of a vertex (# of neighbors)
    - O(v)
- Space?
  - Θ(v + e) memory
  - overhead of node use
  - Could be much less than v^2